Hystrix Python Documentation

Release 0.1.0

Hystrix Python Authors

Oct 31, 2017

Contents

Contents:

CHAPTER 1

Modules

hystrix

hystrix package

Submodules

hystrix.circuitbreaker module

class hystrix.circuitbreaker.**CircuitBreakerMetaclass**
Bases: type

class hystrix.circuitbreaker.**CircuitBreaker**
Bases: object

hystrix.command module

Used to wrap code that will execute potentially risky functionality (typically meaning a service call over the network) with fault and latency tolerance, statistics and performance metrics capture, circuit breaker and bulkhead functionality.

class hystrix.command.**CommandMetaclass**
Bases: type

class hystrix.command.**Command**(*timeout=None*)
Bases: object

run()

fallback()

cache()

execute(*timeout=None*)

observe(*timeout=None*)

queue(*timeout=None*)

hystrix.executor module

class hystrix.executor.**ExecutorMetaclass**
Bases: type

class hystrix.executor.**Executor**(*max_workers=5*)
Bases: concurrent.futures.thread.ThreadPoolExecutor

hystrix.group module

class hystrix.group.**GroupMetaclass**
Bases: type

class hystrix.group.**Group**
Bases: object

hystrix.metrics module

Used by *hystrix.command.Command* to record metrics.

class hystrix.metrics.**Metrics**(*counter*)
Bases: object

Base class for metrics

Parameters **counter** (*hystrix.rolling_number.RollingNumber*) – Used to increment or set values over time.

cumulative_count(*event*)
Cumulative count

Get the **cumulative** count since the start of the application for the given RollingNumberEvent.

Parameters **event** (RollingNumberEvent) – The Event to retrieve a **sum** for.

Returns Returns the long cumulative count.

Return type long

rolling_count(*event*)
Rolling count

Get the rolling count for the given:class:*RollingNumberEvent*.

Parameters **event** (RollingNumberEvent) – The Event to retrieve a **sum** for.

Returns Returns the long cumulative count.

Return type long

class hystrix.metrics.**CommandMetricsMetaclass**
Bases: type

Metaclass for *CommandMetrics*

Return a cached or create the *CommandMetrics* instance for a given *hystrix.command.Command* name.

This ensures only 1 *CommandMetrics* instance per *hystrix.command.Command* name.

class hystrix.metrics.**CommandMetrics**
Bases: *hystrix.metrics.Metrics*

Command metrics

class hystrix.metrics.**ExecutorMetricsMetaclass**
Bases: type

class hystrix.metrics.**ExecutorMetrics**
Bases: object

class hystrix.metrics.**HealthCounts**(*total*, *error*, *error_percentage*)
Bases: object

Number of requests during rolling window.

Number that failed (failure + success + timeout + thread pool rejected + short circuited + semaphore rejected).

Error percentage;

total_requests()
Total reqeust

Returns Returns total request count.

Return type int

error_count()
Error count

Returns Returns error count.

Return type int

error_percentage()
Error percentage

Returns Returns error percentage.

Return type int

hystrix.rolling_number module

class hystrix.rolling_number.**RollingNumber**(*_time*, *milliseconds*, *bucket_numbers*)
Bases: object

A **number** which can be used to track **counters** (increment) or set values over time.

It is *rolling* in the sense that a `milliseconds` is given that you want to track (such as 10 seconds) and then that is broken into **buckets** (defaults to 10) so that the 10 second window doesn't empty out and restart every 10 seconds, but instead every 1 second you have a new *Bucket* added and one dropped so that 9 of the buckets remain and only the newest starts from scratch.

This is done so that the statistics are gathered over a *rolling* 10 second window with data being added/dropped in 1 second intervals (or whatever granularity is defined by the arguments) rather than each 10 second window starting at 0 again.

Performance-wise this class is optimized for writes, not reads. This is done because it expects far higher write volume (thousands/second) than reads (a few per second).

For example, on each read to getSum/getCount it will iterate buckets to sum the data so that on writes we don't need to maintain the overall sum and pay the synchronization cost at each write to ensure the sum is up-to-date when the read can easily iterate each bucket to get the sum when it needs it.

See test module `tests.test_rolling_number` for usage and expected behavior examples.

buckets_size_in_milliseconds()

increment(*event*)

Increment the **counter** in the current bucket by one for the given *RollingNumberEvent* type.

The *RollingNumberEvent* must be a **counter** type

```
>>> RollingNumberEvent.isCounter()
True
```

Parameters **event** (*RollingNumberEvent*) – Event defining which **counter** to increment.

update_rolling_max(*event*, *value*)

Update a value and retain the max value.

The *RollingNumberEvent* must be a **max updater** type

```
>>> RollingNumberEvent.isMaxUpdater()
True
```

Parameters

- **value** (*int*) – Max value to update.
- **event** (*RollingNumberEvent*) – Event defining which **counter** to increment.

current_bucket()

Retrieve the current *Bucket*

Retrieve the latest *Bucket* if the given time is **BEFORE** the end of the **bucket** window, otherwise it returns `None`.

The following needs to be synchronized/locked even with a synchronized/thread-safe data structure such as LinkedBlockingDeque because the logic involves multiple steps to check existence, create an object then insert the object. The 'check' or 'insertion' themselves are thread-safe by themselves but not the aggregate algorithm, thus we put this entire block of logic inside synchronized.

I am using a `multiprocessing.RLock` if/then so that a single thread will get the lock and as soon as one thread gets the lock all others will go the 'else' block and just return the currentBucket until the newBucket is created. This should allow the throughput to be far higher and only slow down 1 thread instead of blocking all of them in each cycle of creating a new bucket based on some testing (and it makes sense that it should as well).

This means the timing won't be exact to the millisecond as to what data ends up in a bucket, but that's acceptable. It's not critical to have exact precision to the millisecond, as long as it's rolling, if we can instead reduce the impact synchronization.

More importantly though it means that the 'if' block within the lock needs to be careful about what it changes that can still be accessed concurrently in the 'else' block since we're not completely synchronizing access.

For example, we can't have a multi-step process to add a bucket, remove a bucket, then update the sum since the 'else' block of code can retrieve the sum while this is all happening. The trade-off is that we don't maintain the rolling sum and let readers just iterate bucket to calculate the sum themselves. This

is an example of favoring write-performance instead of read-performance and how the tryLock versus a synchronized block needs to be accommodated.

Returns Returns the latest *Bucket* or None.

Return type bucket

reset()
Reset all rolling **counters**

Force a reset of all rolling **counters** (clear all **buckets**) so that statistics start being gathered from scratch.

This does NOT reset the *CumulativeSum* values.

rolling_sum(*event*)
Rolling sum

Get the sum of all buckets in the rolling counter for the given *RollingNumberEvent*.

The *RollingNumberEvent* must be a **counter** type

```
>>> RollingNumberEvent.isCounter()
True
```

Parameters **event** (*RollingNumberEvent*) – Event defining which counter to retrieve values from.

Returns

Return value from the given *RollingNumberEvent* counter type.

Return type long

rolling_max(*event*)

values(*event*)

value_of_latest_bucket(*event*)

cumulative_sum(*event*)
Cumulative sum

The cumulative sum of all buckets ever since the start without rolling for the given :class‘RollingNumberEvent‘ type.

See *rolling_sum()* for the rolling sum.

The *RollingNumberEvent* must be a **counter** type

```
>>> RollingNumberEvent.isCounter()
True
```

Parameters **event** (*RollingNumberEvent*) – Event defining which **counter** to increment.

Returns

Returns the cumulative sum of all increments and adds for the given *RollingNumberEvent* counter type.

Return type long

class hystrix.rolling_number.**BucketCircular**(*size*)
Bases: collections.deque

This is a circular array acting as a FIFO queue.

size

last()

peek_last()

add_last(*bucket*)

class hystrix.rolling_number.**Bucket**(*start_time*)
Bases: object

Counters for a given *Bucket* of time

We support both *LongAdder* and *LongMaxUpdater* in a *Bucket* but don't want the memory allocation of all types for each so we only allocate the objects if the *RollingNumberEvent* matches the correct **type** - though we still have the allocation of empty arrays to the given length as we want to keep using the **type** value for fast random access.

get(*event*)

adder(*event*)

max_updater(*event*)

class hystrix.rolling_number.**LongAdder**(*min_value=0*)
Bases: object

increment()

decrement()

sum()

add(*value*)

class hystrix.rolling_number.**LongMaxUpdater**(*min_value=0*)
Bases: object

max()

update(*value*)

class hystrix.rolling_number.**CumulativeSum**
Bases: object

add_bucket(*bucket*)

get(*event*)

adder(*event*)

max_updater(*event*)

hystrix.rolling_number.**RollingNumberEvent**
alias of THREAD_MAX_ACTIVE

hystrix.rolling_percentile module

class hystrix.rolling_percentile.**RollingPercentile**(*_time*, *milliseconds*, *bucket_numbers*, *bucket_data_length*, *enabled*)
Bases: object

buckets_size_in_milliseconds()

current_bucket()

add_value(**values*)
Add value (or values) to current bucket.

percentile(*percentile*)

current_percentile_snapshot()

mean()

class hystrix.rolling_percentile.**Bucket**(*start_time*, *bucket_data_length*)
Bases: object

Counters for a given 'bucket' of time.

class hystrix.rolling_percentile.**PercentileBucketData**(*data_length*)
Bases: object

add_value(**latencies*)

length()

class hystrix.rolling_percentile.**PercentileSnapshot**(**args*)
Bases: object

percentile(*percentile*)

compute_percentile(*percent*)

mean()

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

I

L

M

O

P

Q

R

S

T

U

V